(NASA-CR-199282) AN IMPLEMENTATION AND PERFORMANCE MEASUREMENT OF THE PROGRESSIVE RETRY TECHNIQUE (Bell Telephone Labs.) 8 p

N96-11209

Unclas

G3/61 0065036

An Implementation and Performance Measurement of the Progressive Retry Technique

Gaurav Suri * Yennun Huang * Yi-Min Wang* W. Kent Fuchs† Chandra Kintala*

*AT&T Bell Laboratories
600 Mountain Avenue
Murray Hill, NJ 07974

†Coordinated Science Laboratory
University of Illinois
Urbana, IL 61801

Abstract

This paper describes a recovery technique called progressive retry for bypassing software faults in message-passing applications. The technique is implemented as reusable modules to provide application-level software fault tolerance. The paper describes the implementation of the technique and presents results from the application of progressive retry to two telecommunications systems. The results presented show that the technique is helpful in reducing the total recovery time for message-passing applications.

1 Introduction

For computer systems designed to provide continuous services to customers, availability is an important performance measure. In such systems, software failures have been observed to be the current major cause of service unavailability [1, 2]. Residual software faults due to untested boundary conditions, unanticipated exceptions and unexpected execution environments have been observed to escape the testing and debugging process and, when triggered during program execution, cause service interruption [3]. It is therefore desirable to have effective on-line retry mechanisms for automatically bypassing software faults and recovering from software failures in order to achieve high availability [4, 5, 6, 7].

Several studies [2, 8, 9] have shown that many software failures in production systems behave in a transient fashion, and so the simplest way to recover from such failures is to restart the system, an approach that we call *environment diversity*. The term *Heisenbug* [1] has been used to refer to the software faults causing transient failures, while the term *Bohrbug* refers to software faults which have deterministic behavior.

`Watchd` daemon and `libft` library have been used in several AT&T products to tolerate Heisenbugs [10]. `Watchd` is a daemon process which monitors system failures like machine crash, process death and process hang. If a machine crashes, all critical applications running on the crashed machine are migrated to another machine. If a process dies (due to bugs in the program), `watchd` first restarts the process locally. If the restarted process fails again, the process is then migrated to another machine. `Libft` provides functions for message logging, critical-data checkpointing, fault-tolerant inter-process communication and name services. With `libft`, an application process can checkpoint its critical data on the local machine as well as on backup machines. Therefore, when a process is restarted, it can restore its checkpointed state and replay its message log to reconstruct its pre-failure state. `Watchd` keeps track of the dependence between processes. In the event of a failure, the failed process as well as all other processes that depend on it are rolled back in order to guarantee state consistency. `Watchd` and `libft` together provides a simple, portable and reusable component for an application to tolerate Heisenbugs.

Our experience has shown that many errors can be successfully tolerated by using the simple rollback-and-retry mechanism provided by `watchd` and `libft`. The simple mechanism, although effective, presents some problems. First, the recovery time can be long so that the service disruption due to the recovery can be unbearable. Any process failure results in a global restart. In an application consisting of many processes, a global restart can take a long time before the application returns to normal execution. Therefore, it is desirable to limit the scope of rollback by keeping track of the dynamic inter-process communication patterns and rolling back only the processes which directly communicate with the failed processes in the current checkpoint interval. Second, the simple retry with a deterministic replay of message logs usually reconstructs the application state to the same state as existed before failure. If the state is erroneous and the application behavior is deterministic, the retry and replay will not help. However, if message dependency is recorded, a failed application can replay the messages in a different but consistent order so that the appli-

The research of the fourth author was supported in part by the Department of the Navy and managed by the Office of the Chief of Naval Research under Contract N00014-91-J-1283, and in part by the National Aeronautics and Space Administration (NASA) under Grant NASA NAG 1-613, in cooperation with the Illinois Computer Laboratory for Aerospace Systems and Software (ICLASS).

0-8186-7059-2/95 $04.00 © 1995 IEEE

cation reaches a new but correct state after retry. In other words, by replaying the message log in a different but consistent order, more software failures may be bypassed.

The above two observations motivate the extension of the rollback-and-retry mechanism provided by `watchd` and `libft`. This paper describes a *progressive retry* technique for software failure recovery in message-passing applications[1]. The target applications are continuously-running software systems for which fast recovery is essential and a reasonable amount of run-time overhead may not result in noticeable service quality degradation. Many telecommunications systems fall into this category. There are several reasons that fast recovery is desirable in applications requiring high availability. In the cases where service quality is judged at the user interface level, small "computer down time" involving only a small number of processes may be translated into zero "service down time." Most importantly, when the prolonged unavailability of one part of the system may trigger the boundary conditions in other parts of the system, localized and fast recovery can reduce the possibility of cascading failures which may lead to a catastrophe.

The progressive retry technique is based on checkpointing, rollback, message replaying and message reordering. The goal is to limit the *scope of rollback*: the number of involved processes as well as total rollback distance. The approach consists of several retry steps and gradually increases the scope of rollback when a previous retry fails. The technique is implemented in `watchd` daemon and `libft` library.

2 Progressive Retry

The simple example in Fig. 1 is used to illustrate the basic concept of progressive retry. The reader is referred to [12] for a detailed discussion. For the purpose of presentation, we assume every message is logged before it is processed, and is therefore available at the time of recovery. Suppose p_2 detects an error at the point marked "X" in Figure 1 and initiates the progressive retry. In the Step-1 *receiver replaying retry*, p_2 rolls back and replays messages M_a and M_b in exactly the same order as they were processed before the rollback. If the detected error was caused by some transient environmental problems (such as mutual exclusion conflicts, resource unavailability, unexpected signals, etc.) then Step-1 retry may succeed and p_2 can proceed. Under the deterministic assumption, the exact same copy of M_o will be generated during the recovery. Therefore, p_2 does not need to resend M_o and the receiver of M_o, process p_4, does not have to be involved in the retry.

If Step-1 retry fails, then p_2 rolls back again and executes Step-2 *receiver reordering retry* by reordering M_a and M_b in its message log. If the original error was triggered by a boundary condition, then message reordering may be useful for bypassing that condition and thereby recovering from the error. Note that since

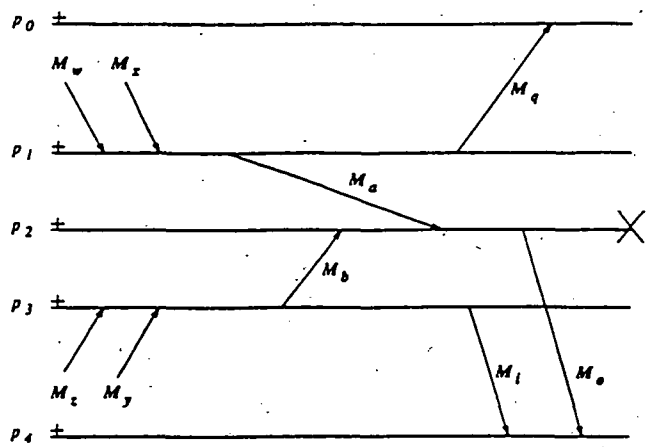


Figure 1: Example for illustrating the basic concept of progressive retry.

message reordering forces a different execution path for p_2, we cannot expect that the same message M_o will still be generated. Such a message is called an *orphan* message [11] and should be discarded. As a result, p_4 should also be rolled back in order to undo the effect of M_o. The message M_i, however, is not an orphan message because its sender is not rolled back. Such a "sent but not yet received message" is called an *in-transit message* [12]. It needs to be processed again by the restarted receiver but its associated processing order information can be discarded.

There are several potentially useful algorithms for reordering the message logs. Random reordering can be used when no knowledge about the possible cause of the software failure is available. If the failure is possibly due to the interleaving of messages from different processes, reordering by grouping the messages from the same process together may be useful. If the software fault might have been triggered by exhausting all available resources, reordering the messages so that every resource is freed at the earliest possible moment can often bypass the boundary condition.

If Step-2 retry fails, then Step-3 *sender replaying retry* will involve in the recovery process all the processes that have sent messages to p_2. In Fig. 1, p_1 (p_3) rolls back and replays M_w and M_x (M_z and M_y) in their original order[2]. Step-3 retry basically gives the messages a second chance to interleave "naturally" with each other, and can be useful for error recovery if the original error was due to some rare message racing conditions. Under the deterministic assumption, the exact same copy of M_q will be generated and so p_0 does not need to be involved in the rollback. In contrast, p_4 needs to roll back because of the orphan message M_o, and M_i remains an in-transit message.

If Step-3 retry still fails, it is suspected that an undetected error might have occurred at p_1 or p_3 and was propagated to p_2 through the erroneous messages

[1] We will focus on error recovery in this paper; the issue of error detection is considered elsewhere [10].

[2] Messages M_w, M_x, M_y and M_z are from processes other than the five processes shown in the Figure.

M_a or M_b to cause the detected error. Step-4 *sender reordering retry* is designed to bypass the software bug that caused the undetected error. In this step, process p_1 rolls back and reorders M_w and M_x, and p_3 rolls back and reorders M_z and M_y. As a result, messages M_a, M_b, M_i, M_o and M_q all become orphan messages and all the five processes in Fig. 1 need to participate in the recovery.

When all previous small-scope retries fail, the objective of localized recovery can no longer be achieved and a large-scope rollback needs to be initiated. All the processes in the system, including the senders of M_w, M_x, M_y and M_z, are rolled back to the latest globally consistent set of checkpoints obtained through coordinated checkpointing [13] or lazy coordination [14]. If a system has been functioning correctly for most of the time and failures are rare events, rolling back the entire system can often recover from the failures. The potential disadvantages of a large-scope rollback include unnecessarily involving healthy critical processes in the rollback and a longer recovery time. In a later section, we show that, for certain systems, the cost of the first four steps of progressive retry is small compared to the cost of a large-scope rollback. For such systems, the five-step progressive retry described in this section is an attractive technique for providing low-cost and efficient software failure recovery.

3 Implementation

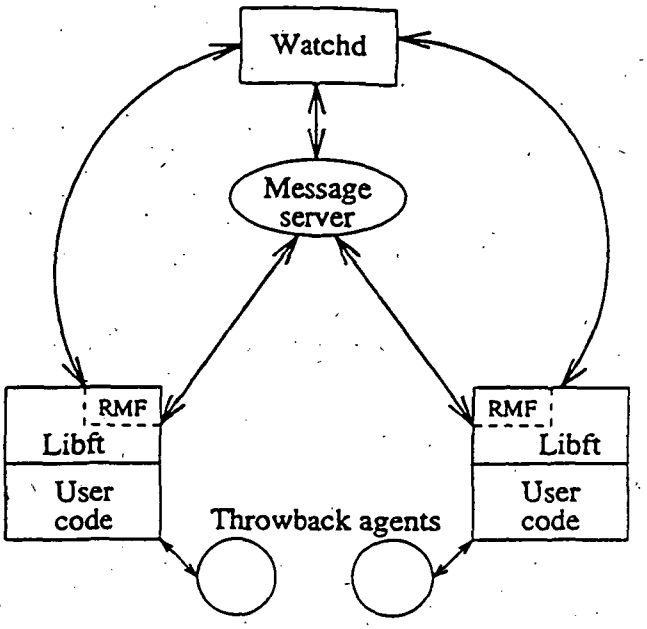
The *progressive retry* mechanism is implemented in the *libft* library and the *watchd* daemon [10]. The heart of the system is a centralized *message server* which dynamically keeps track of all the information required for progressive retry. The message server runs as a child of the watchd daemon and uses the checkpointing capabilities of the libft library in order to make its own operation fault-tolerant. The other important components of the progressive retry mechanism are *watchd*, the *throwback agents* and the *recovery management functions* (see Figure 2).

Watchd monitors user processes for failures. As soon as it detects a failure, it restarts the process. It also communicates with the message server to get information about the other processes that need to be restarted. It then kills and restarts those processes. The first action taken by each of the restarted processes is to communicate with the message server and find out the recovery actions that need to be taken. Each process then sets up its recovery status accordingly and proceeds with recovery. The following subsections explain each of these functions in detail.

3.1 Message Server

As described earlier, the message server is the most important component of the progressive retry mechanism. It has the following functions :

- Keep track of the communication graph during failure-free operation,
- Maintain status information for each process in the system, and



RMF - Recovery Management Functions

Figure 2: Progressive retry system architecture

- Compute the recovery line during failure recovery.

3.1.1 Dynamic Communication Graph

The message server needs to keep track of the message dependencies during normal program execution in order to be able to compute the recovery line during the recovery process. The graph is computed on the basis of the pattern sent to the message server by the receivers of messages in the system. Each process maintains a local communication graph in which it keeps track of all the processes that have sent messages to it so far. Whenever it gets a message from a process that is not present in its local graph, it adds the process to the local graph and also sends the information to the message server so that the global communication graph can be updated.

3.1.2 Process Status Information

When the application is recovering from the failure of a process under progressive retry, the status of other processes is also affected to some degree depending on the communication pattern and the stage of retry the system is in. The processes need to be assigned different status values so that they know whether they have to deterministically replay the pre-failure receiver log, reorder and replay the log, receive in-transit messages from the communication channel or perform as normal processes. The reasons for making these distinctions are explained in Section 3.4.

3.1.3 Recovery Line Computation

Recovery line computation is the first step to be carried out when a failed process restarts and progressive retry needs to be initiated. It involves carrying out the following functions:

- Calculate the retry step number for the system
- Analyze the communication graph, and
- Determine the status of each affected process based on the first two steps

The recovery line computation may result in a change of status for some of the processes in the application. Since all processes run as children of watchd, this information is conveyed to watchd so that it can take appropriate action and restart processes that need a status change.

3.2 Watchd

The basic functions of watchd [10] are to periodically monitor processes to see if they are alive, and to restart failed processes. Under progressive retry watchd is given the following additional responsibilities:

- Invoke the message server to initiate progressive retry when a failed process is brought up again.
- Kill and restart all the processes that require a status change during retry.

3.3 Throwback Agents

The *throwback agents* are invoked on a one-per-process basis and their function is to simulate the presence of *in-transit* messages. If a process is assigned a status which implies that there are pending in-transit messages, these messages need to be resent to it during recovery. The process fires a throwback agent which analyzes the log of the process and sends back to the process all messages that have become in-transit according to the new recovery line. Once all in-transit messages have been re-sent, the throwback agent terminates indicating the completion of recovery at that node.

3.4 Recovery Management Functions

The recovery management functions are a part of *libft* and are responsible for recovery initialization, setup and management for each process in a local manner. The functions in libft that do recovery management are `checkpoint()`, `recover()`, `recovered()`, `setlogfile()`, `ftrecsetup()`, `ftread()` and `ftwrite()`.

`ftrecsetup()` is the function that sets up the recovery for each process. It communicates with the message server to get the recovery line information. It then uses that information to set its local status value and fire a throwback agent, if required.

The function `recovered()` returns a value which indicates which stage of recovery the system is in. The stages can be: doing deterministic replay, receiving in-transit messages, or recovery completed. The return value is used by the process to determine whether it should be receiving messages from the communication channel or retrieving them from the log files. These values, in conjunction with the status values are also used by `ftread()` and `ftwrite()`. The function `ftread()` examines the status value of the process, and based on that value reads the next message either from the log or from the communication channel. Even when reading from the channel, a distinction is made for receiving in-transit messages and new messages. In order to maintain consistency, all in-transit messages from a sender must be received before any new messages can be received from that sender and *fifo* order maintained for the in-transit messages. The received messages are logged before they can be processed. The status value determines whether they are logged in a temporary log file or in the regular log file.

The function `ftwrite()` sends messages after logging them. The status values indicate whether message comparison needs to be done (in order to verify the deterministic execution assumption) or not, and whether the message actually needs to be sent out on the communication channel at all.

A message in a receiver log file contains five fields: message sequence number, sender id, reference id, message size and message data. Sender id is the number assigned by watchd to each application at start-up time. Sequence number is used during message reordering to ensure that *fifo* order for messages from the same sender is maintained. Reference id is given by `ftwrite()` and is also used during message reordering. The message structure in the sender log is the same except that it contains the receiver id instead of sender id, and it does not contain the reference id.

4 Experimental Results

Performance measurements for failure-free overhead and recovery time were carried out by applying progressive retry to two telecommunications systems. The two systems used were the REPL [15] file system, and a subsystem of a switched service network system (which we refer to only as System N due to its proprietary nature).

Since most of the source code was unavailable and our objective was to measure the performance, not the effectiveness, we implemented two simulators which use the same software architectures, follow the same communication patterns and generate the same workload conditions as the actual systems. The simulators were also useful for doing controlled fault injection at specific points in the programs.

4.1 General Experimental Setup

The experiments for both the systems studied the performance under two categories :

- measurement of failure-free overhead;
- measurement of recovery time for different steps of progressive retry.

The run time of each simulation was in the order of one hour or more and each measurement was averaged over four runs.

4.2 Performance measurement on System N

The part of System N that we modeled has the communication pattern shown in Figure 3. Nodes A and B report to Node *D* every 10 seconds. *C* sends a status report to *D* every 2 seconds. *D* periodically reports to *F* which also receives reports from *E* every 30 seconds and which in turn reports to *G* [3].

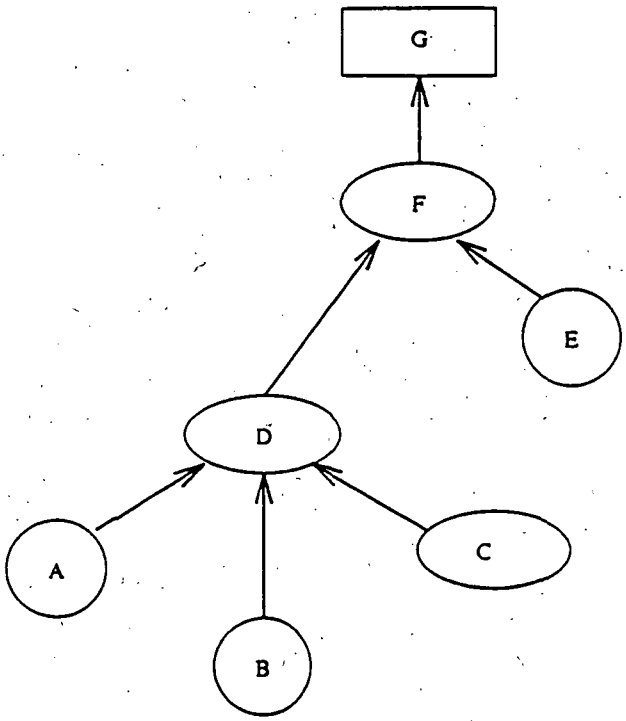


Figure 3: Communication pattern for the part of System N under study

System N uses a coordinated checkpointing scheme where process *D* is the coordinator. The failure free overhead was measured for two different checkpoint intervals: a checkpoint every 200 messages received by *D* and every 400 messages. Synchronous logging was used for both sender and receiver logging. The critical data sizes for each of the processes were of the order of a few kilobytes.

The recovery time measurements were done for the checkpoint interval with 400 messages in order to get a worst case measure of timing. Three different failure instants were assumed: failure at 25% of checkpoint interval, at 50% of checkpoint interval and at 75% of checkpoint interval. The failures were injected at the node shown as *G* in the Figure.

The observations are given in Tables 1 and 2. Table 1 shows the failure-free overhead for System N while Table 2 shows actual recovery time in seconds for step 1 alone, steps 1 & 2, steps 1, 2 & 3, steps 1, 2, 3 & 4, and step 5. Table 2 also shows the recovery times for the first four cases as a percentage of the time taken for step 5 (large-scope rollback recovery) and the number of processes involved at each step.

[3]The timings used in the simulations were obtained from the specification documents of the system.

Table 1: Failure-free overhead for System N

Execution Type	No logging/ checkpointing	Chk&Logging 400	Chk&Logging 200
Time(s)	3040	3137	3140
% overhead	-	3.1%	3.2%

From Table 1, the run time overhead of message logging and checkpointing for system N is about 3%. In case of a failure, the time taken for doing retry is very low compared to the time that large-scale rollback takes, as shown in Table 2. Also note that the number of processes involved in doing steps 1 to 4 is at most 2, compared to 7, which is the number of processes involved in step 5. In most systems, the smaller the number of processes involved in recovery, the less impact the failure has. The results shown here coupled with this observation make the progressive retry technique extremely attractive for use with System N.

System N has been deployed in the field for more than 2 years now. Data obtained from the field has shown that more than 90% of the exceptions (failures) that occurred in the last 2 years have been successfully recovered by steps 1 to 3.

4.3 Performance measurement on REPL

REPL [15] is a collection of file system library functions and server processes that runs on a primary and a backup machine. Applications run on the primary machine and write critical files onto the primary file system. The REPL library intercepts the file system calls, produces update messages and sends the update messages to the REPL server processes, which then transfer the update messages to the REPL processes on the backup node. The backup REPL processes replay the update messages and reproduce the file updates on the backup node. REPL has been used in several telecommunications systems to replicate critical files and databases. The communication graph for REPL is shown in Figure 4.

A normal workload for REPL is a burst mode workload. It receives a burst of messages from an application, then becomes idle for some time and this cycle repeats over and over again. Since the burst frequency depends on the application that is using REPL, it is not possible to define one workload for the system. Thus the experiment has to be conducted for different burst frequencies.

A standard burst size of 10 messages per burst was used for the experiment. The workload was varied between 6 bursts per minute and 1.25 bursts per minute for measuring the recovery time. The failure free overhead was measured for the workload with a frequency of 6 standard bursts per minute, which is the worst

Table 2: Recovery times for System N. t:time in seconds, %:recovery time as a percentage of step 5 time, n:number of processes involved

Failure at	Step 1			Steps 1,2			Steps 1-3			Steps 1-4			Step 5	
	t	%	n	t	%	n	t	%	n	t	%	n	t	n
25%-chk	2s	1.4%	1	7s	4.8%	1	16s	11.0%	2	26s	17.9%	2	145s	7
50%-chk	3s	1.1%	1	8s	2.8%	1	18s	6.3%	2	28s	9.8%	2	285s	7
75%-chk	4s	0.9%	1	9s	2.1%	1	20s	4.6%	2	30s	6.9%	2	437s	7

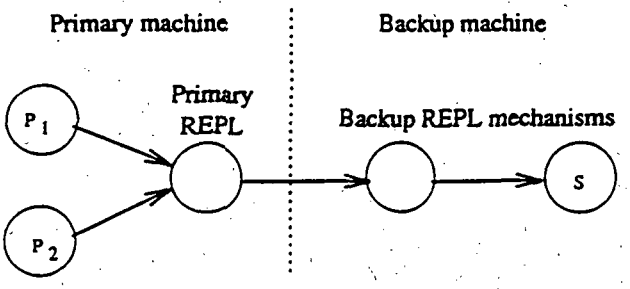


Figure 4: Communication pattern for REPL

case workload due to the high communication frequency.

The failure-free overhead measurements are given in Table 3. Checkpoint intervals of 500 messages, 400 messages and 250 messages per checkpoint were studied. The checkpoint size varied from 10 Kb to about 100Kb. The failure-free overhead for the worst case workload has a maximum value of 10.9%, which is acceptable in most REPL applications.

Table 3: Failure-free overhead for REPL

Execution Type	No logging/ checkpointing	Chk&Logging		
		500	400	250
Time(s)	3092	3387	3398	3432
% overhead	-	9.5%	9.8%	10.9%

Recovery time data was collected for message densities of 6, 3, 2, 1.5 and 1.25 standard bursts per minute (see Figures 5 and 6). For each message density the data was collected for failure rates corresponding to failures at 20% of checkpoint interval, 40% of checkpoint interval, 60% of checkpoint interval and 80% of checkpoint interval. Fault injection was done at the node marked S in the Figure. Figures 5 and 6 present the plots corresponding to failures at 20% and 80% of the checkpoint interval.

From the Figures, we observe that:

- steps 1 and 2 have a low recovery time compared to that of step 5 for all the message densities; therefore, steps 1 and 2 are very attractive for various message densities;
- step 3 or step 4 of progressive retry could save a lot of recovery time only if the message density is low;
- the later a failure occurs in a checkpoint interval, the lower is the percentage of the recovery time compared with that of the step 5 retry. In other words, if a failure occurs later in a checkpoint interval of a system, progressive retry has a greater impact in reducing the recovery time provided the system successfully recovers at an early step.

5 Concluding Remarks

We have described a 5-step progressive retry technique using message logging as well as checkpointing to limit the scope of rollback and thereby provide a means for achieving localized and fast recovery. The technique is designed for continuously-running software systems which can absorb a certain degree of performance overhead and significantly benefit from reduced service unavailability. Our approach, which is based on the piecewise deterministic execution model, employs message replay to reconstruct state during recovery, message comparison to verify whether the above assumption is true, and message reordering to introduce environment diversity. Progressive retry has been implemented as part of a Software Fault Tolerance Platform developed at AT&T Bell Laboratories to provide automatic, economical, effective and efficient software failure recovery. Experiments conducted using this implementation of progressive retry have shown that the technique can significantly reduce failure recovery time while incurring only small performance overhead. Experience has also shown that incorporating progressive retry is easy as it requires adding only a few lines of code to a program.

Acknowledgements

The authors wish to express their sincere thanks to Chia-Mei Chen for her contribution to the implementation.

References

[1] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*. San Mateo, CA: Morgan Kaufmann Publishers, 1993.

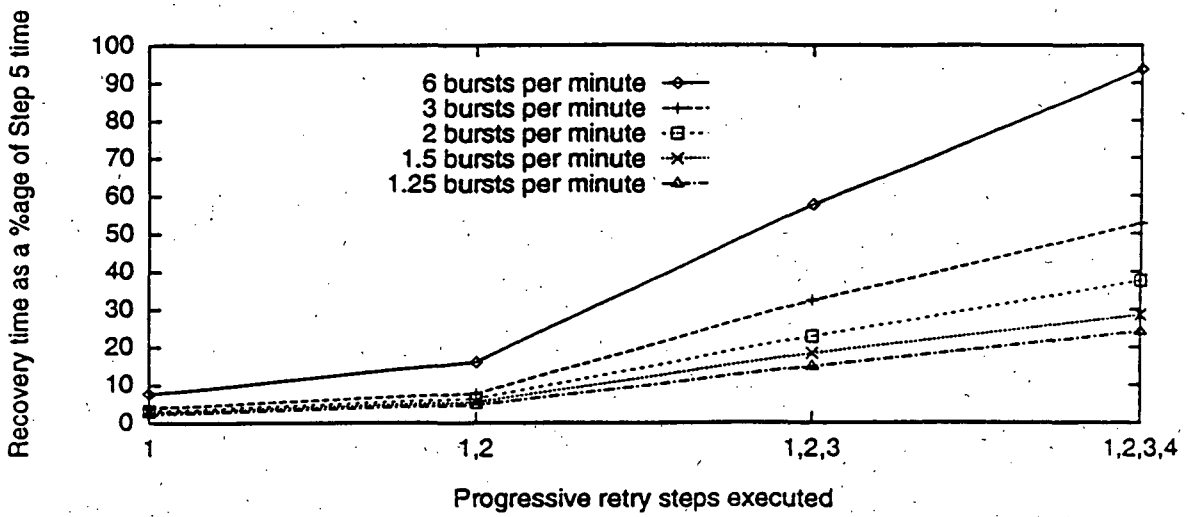


Figure 5: REPL : Time taken for retry (as a percentage of large-scope rollback recovery time) vs. the retry steps executed: failure at 80% checkpoint interval

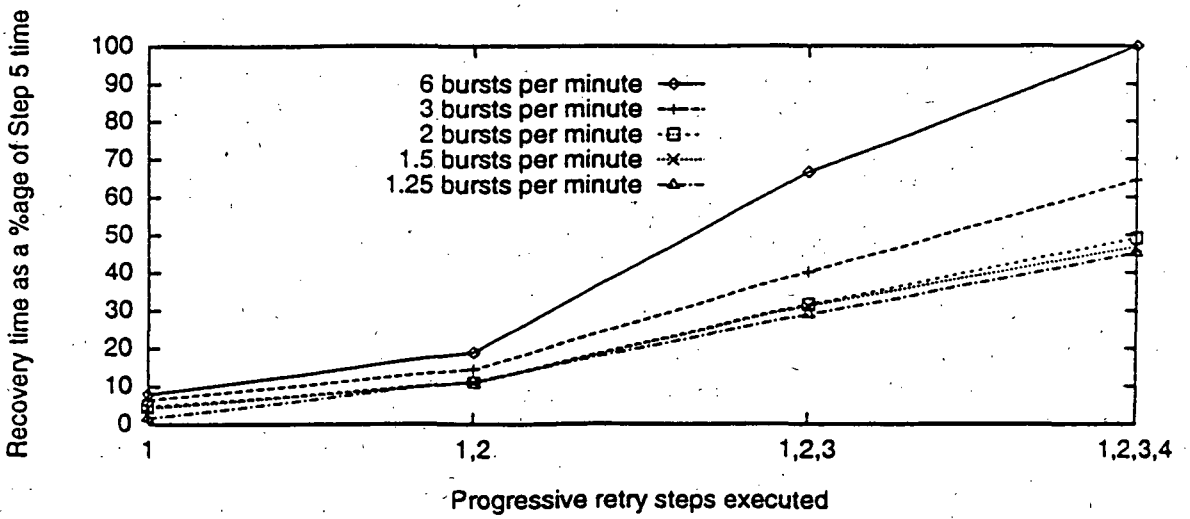


Figure 6: REPL : Time taken for retry (as a percentage of large-scope rollback recovery time) vs. the retry steps executed: failure at 20% checkpoint interval.

[2] J. Gray, "A census of tandem system availability between 1985 and 1990," *IEEE Trans. Reliab.*, Vol. 39, No. 4, pp. 409–418, Oct. 1990.

[3] M. Sullivan and R. Chillarege, "Software defects and their impact on system availability - A study of field failures in operating systems," in *Proc. IEEE Fault-Tolerant Computing Symp.*, pp. 2–9, 1991.

[4] D. Jewett, "Integrity S2: A fault-tolerant UNIX platform," in *Proc. IEEE Fault-Tolerant Computing Symp.*, pp. 512–519, 1991.

[5] J. Gray and D. P. Siewiorek, "High-availability computer systems," *IEEE Comput. Mag.*, pp. 39–48, Sept. 1991.

[6] J. Gray, "Dependable systems." *Keynote Speech, 11th Symp. on Reliable Distr. Syst.*, Oct. 1992.

[7] F. Cristian, "Exception handling and software fault tolerance," *IEEE Trans. Comput.*, Vol. C-31, No. 6, pp. 531–540, June 1982.

[8] E. Adams, "Optimizing preventive service of software products," *IBM J. R&D*, No. 1, pp. 2–14, Jan. 1984.

[9] I. Lee and R. K. Iyer, "Faults, symptoms, and software fault tolerance in the tandem guardian90 operating system," in *Proc. IEEE Fault-Tolerant Computing Symp.*, 1993.

[10] Y. Huang and C. Kintala, "Software implemented fault tolerance: Technologies and experience," in *Proc. IEEE Fault-Tolerant Computing Symp.*, pp. 2–9, June 1993.

[11] R. E. Strom and S. Yemini, "Optimistic recovery in distributed systems," *ACM Trans. Comput. Syst.*, Vol. 3, No. 3, pp. 204–226, Aug. 1985.

[12] Y. M. Wang, Y. Huang, and W. K. Fuchs, "Progressive retry for software error recovery in distributed systems," in *Proc. IEEE Fault-Tolerant Computing Symp.*, pp. 138–144, June 1993.

[13] K. M. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed systems," *ACM Trans. Comput. Syst.*, Vol. 3, No. 1, pp. 63–75, Feb. 1985.

[14] Y. M. Wang and W. K. Fuchs, "Lazy checkpoint coordination for bounding rollback propagation," in *Proc. IEEE Symp. Reliable Distributed Syst.*, pp. 78–85, Oct. 1993.

[15] D. Korn, Y. Huang, G. Fowler, and H. Rao, "A user-level replicated file system," in *Proc. Summer '93 USENIX*, pp. 279–290, June 1993.